University of Southern California
Department of Electrical Engineering
EE557 Fall 2001
Instructor: Michel Dubois
Homework #3. SOLUTIONS

Problem 1 (20pts).

There are seven dependences in the C loop presented in the exercise:

1. True dependence from S1 to S2 on a.
2. Loop-carried true dependence from S4 to S1 on b.
3. Loop-carried true dependence from S4 to S3 on b.
4. Loop-carried true dependence from S4 to S4 on b.
5. Loop-carried output dependence from S1 to S3 on a.
6. Loop-carried antidependence from S1 to S3 on a.
7. Loop-carried antidependence from S2 to S3 on a.

For a loop to be parallel, each iteration must be independent of all others, which is not the case in the code used for this exercise. Because dependences 1, 2, 3 and 4 are "true" dependences, they can not be removed through renaming. In addition, as dependences 2, 3 and 4 are loop-carried, they imply that interatoins of the loop are not independent. These factors together imply the loop can not be made parallel as the loop is written. By rewriting the loop it may be possible to find a loop that is functionally equivalent to the original loop that can be made parallel.

Problem 2 (20pts).

There are six dependences in the C loop presented in the exercise:

1. True dependence from S1 to S2 on a.
2. True dependence from S1 to S3 on a.
3. Anti dependence from S1 to S2 on b.
4. Loop-carried true dependence from S3 to S2 on a.
5. Loop-carried true dependence from S3 to S3 on a.
6. Loop-carried output dependence from S3 to S1 on a.

To parallelize the loop, we simply 'break" all of the loop-carried dependences. The loop-carried output dependence can be removed by the compiler through renaming while the loop-carried true dependences can be removed if the compiler modifies the code.

The C loop in this exercise can be re-written as follows to make it parallel. In this case, this can be accomplished by removing statement S3 from the loop. Looking at the original code, it should be apparent that S3 does not do any useful work in the loop. Although S3 assigns a value to a[i+1], this value is over-written by S1 on the next iteration before the value written by S3 in used. The

transformed version of the loop is functionally equivalent to the original loop from the exercise.

```
for (i=1; i<100; i=i+1) {
a[i]=b[i]+c[i] ; /* S1*/
b[i]=a[i]+d[i] ; /*S2 */
}
a[100]=a[99]+e[99] ;
```

Problem 3 (20pts).

All the statements inside the THEN and ELSE parts of the code are lexically control dependent upon the conditional "if (a<c)" since they are executed conditionally based on the result of the comparison (some of these control dependences can be ignored due to an absence of destructive data dependences, such cases are described in the list below).

1. d=d+5; Can be scheduled before the conditional since none of the data dependences between d and any of the code in the conditional, the branches of the conditional, or the code after the conditional are affected.
2. a=b+d+e; Can not be moved since the value of a is changed conditionally and its value is used after the body of the conditional.
3. e=e+2; Can not be moved since the value of a is computed using the old value of e in the THEN part of the conditional.
4. f=f+2; Can not be moved since the value of f is conditionally computed and the value of f is used after the conditional statement.
5. c=c+f; Can not be moved since it depends on the value of f computed by the prior statement and the value of c is needed beyond the shown code segment (as specified in the exercise statement).
6. b=a+f; This statement can not be moved because it is data dependent on values computed in the conditional statement.

These answers are based on the assumption that no variable renaming is performed.

Problem 4 (20pts).

This question can not be easily solved using only unrolling. The solution presented here has two stall cycles after the loop body, but the loop itself is stall-free.

For the solution below, we assume that this loop will be executed a nonzero, even number of times. We have also scheduled the branch-delay slot and follow the same basic rules to unroll the loop. Due to the latencies of the FPU, and small number of instructions in this loop (especially after removing extra SUBI's), unrolling is not effective all by itself.

The solution below takes advantage of the associativity of sum reduction and computes all the even indices into one register and all the odd indices into another. By doing this, the instructions from two consecutive iterations of the original loop can be freely scheduled together. After the loop body, the registers holding the even and odd streams are added to get the final result. We assume that registers F2 and F10 are zero prior to beginning the loop. The comments attached to

the code indicate the amount of latency being covered and the source of the latency (e.g., “3 from MULTD F0, F0, F4” indicates that three cycles of latency are being covered from the indicated multiply)

```
loop:
        LD F0, 0(R1)
        LD F6, -8(R1)
        LD F4, 0(R2)
        LD F8, -8(R2)
        MULTD F0, F0, F4 ; 1 from “LD F4, 0(R2)”
        MULTD F6, F6, F8 ; 1 from “LD F8, 8(R2)”
        SUBI R1, R1, 16
        SUBI R2, R2, 16
        ADDD F2, F0, F2 ; 3 from “MULTD F0, F0, F4”

        BNEQZ R1, loop
        ADDD F10, F6, F10 ; 3 from “MULTD F6, F6, F8”
                          ; fill branch delay
out_of_loop:
        ADDD F2, F10, F2 ; combine even-odd streams
```

Problem 5 (100pts).

This question is on the scoreboard processor covered in class. Answer the following questions.
1. Look at Fig. 4.7 page 250 in PH. There may be a subtle problem with WAR hazards. Consider an instruction I in the Read Operand stage waiting for operand Fj while its operand Fk is available (i.e. Rj=0 but Rk=1). Assume that Fj is produced by an instruction preceeding I and currently in FU1. Now assume that the instruction following I modifies Fj; assume that this instruction can execute and completes before the instruction in FU1. The instruction following I will see Rj =0 and thus will not wait in “Write results”. When the instruction in FU1 completes, instruction I gets its operand correctly, but the instruction completing in FU1 overwrites the value of Fj written by the instruction following I. Thus the value in register Fj is now stale.
Explain why this cannot happen.

Answer: This cannot happen because that would imply that two instructions modifying the same register have been issued and are concurrently in progress. The scoreboard solves this problem by the way its prevents WAW hazards: an instruction cannot issue if a previous instruction modifying the same register is pending in the processor.

2. We propose to add register renaming to the scoreboard processor. Instead of 32 physical registers we have now 64 physical registers. The issue logic maintains a table with 32 entries, one for each floating point register in the instructions. Each entry of this table contains 6 bits to point to one physical register. Every time an instruction modifying a register --say register Fj-- goes through the issue stage, the issue stage allocates one physical register --say register Pk-- to Fj and then modifies the destination field of the instruction to point to Pk, so that Pk will now become the physical destination of the instruction. As following instructions reading Fj are issued the

issue stage also changes the operand field of these instructions to Pk, so that the correct value is read.
Each pointer to a physical register also contains one counter. The counter for Fj is incremented everytime an instruction reading Fj goes through the issue logic and is decremented everytime an instruction reading Fj goes through the Write result stage. When the counter reaches zero, then the value in Pk has been read by all instructions that needed it and Pk can be reallocated to another register Fi. When the counters of all the physical registers are non-zero and an instruction needs a new one, the instruction remains in the issue logic until one physical register is free.

As it is this scheme will not work. What should be done to make it work??

Answer: This will not work. First of all, the counters must be associated with the physical registers since an architectural register may have multiple pending values at the same time. Second, the value assigned to an architectural register must remain valid and accessible at least until a following instruction assigns a new value to it, whether the value is read or not by following instructions. So, we set the counter to 1 when the physical register is allocated and we decrement the counter when the architectural register is remapped to a new physical register (i.e. a new value is assigned with the architectural register).

The rest of the algorithm to mange the counters and the deallocation of physical register remains unchanged.

3.With this addition, specify the functionality of a new scoreboard, similar to the table in Fig. 4.7 of PH for this new processor. Note that register renaming has eliminated WAR and WAW hazards and that the scoreboard must now manage the allocation/deallocation of physical registers. Thus the new table may be quite different from the one in Fig. 4.7.

Table 1:

Instruction Status	Wait Until	Bookeeping
Issue	Not_Busy[FU]	Busy[FU]<-yes;Op [FU] <-Op; Fi[FU] <-D; Qj <- Result[S1]; Qk<-Result[S2] Rj<-not Qj; Rk <- not Qk; Result[D]<-D
Read Operands	Rj and Rk	Rj<-no; Rk <-no; Qj<-0;Qk<-0
Execution complete	Functional Unit Done	
Write Results	-----	For all f (if Qj[f]=FU then Rj[f]<-yes) For all f (if Qk[f]=FU then Rk[f]<-yes) Result[Fi[FU]] <-0;Busy[FU]<-0

Problem 6 (100pts).

Problem 4.14 parts (a) (c) and (d) in PH. Please assume the following modifications:
1. In all cases assume that functional units are not pipelined, but that there are enough functional units to avoid any stalls due to structural hazards on functional units.

2. Use the latencies shown in Figure 4.63 for the 5-stage DLX. In the case of scoreboard and Tomasulo these latencies mean that the execution time of a FPMPY is 7 clocks, that the execution time of FPADD is 5 clocks and that any integer instruction execute in one clock.
3. For the classic 5-stage DLX assume that all functional units are fully pipelined and bypassed (forwarded). If contention might exist for the WB stage for two consecutive then the second instruction is not issued until it is known in the issue logic that the contention will not happen.
4. For the scoreboard DLX, use the design specified in class (however, now the number of FUs is infinite, the number of entries in the window is also large enough.
5. For the Tomasulo DLX use enough reservation stations so that structural stalls for these resources do not occur. Use the design as specified in the class, with an infinite number of FUs.
6. In all cases, show the schedule by filling the instruction status table as was used in class. Each entry in the table shows the clock cycle when a given instruction went through a specific phase of execution. Show the content of the table in the clock cycle when the sgti instruction writes its results.
7. In all cases give the execution time from T=0 to the cycle when sgti writes its result.
8. To show the schedule, you simply have to make 3 copies of the table in Figure 2 and fill the table for each of the 3 processors (do not show all the status tables as asked in the problem statement of the book).

SEE SOLUTIONS on Figure 1

Figure 1.1. 5-stage DLX

Table 1:

Instruction	Issue	Exec. complete	Write Results
LD F2,0(R1)	2	3	5
MULTD F4,F2,F0	4	11	13
LD F6, 0(R2)	5	6	8
ADDD F6,F4,F6	11	16	18
SD 0(R2),F6	12	16	18
ADDI R1,R1,#8	16	17	19
ADDI R2,R2,#8	17	18	20
SGTI R3,R1,done	18	18	21

Execution Time: 21 clocks

Figure 2.2. scoreboard DLX

Table 2:

Instruction	Issue	Op Read (scoreboard only)	Exec. complete	Write Results
LD F2,0(R1)	1	2	3	4
MULTD F4,F2,F0	2	5	12	13
LD F6, 0(R2)	3	4	5	6
ADDD F6,F4,F6	7	14	19	20
SD 0(R2),F6	8	21	22	23
ADDI R1,R1,#8	9	10	11	12
ADDI R2,R2,#8	10	11	12	22
SGTI R3,R1,done	11	13	14	15

Execution Time: 15 clocks

Figure 1.2. Tomasulo DLX

Table 3:

Instruction	Issue	Exec. complete	Write Results
LD F2,0(R1)	1	2	3
MULTD F4,F2,F0	2	10	11
LD F6, 0(R2)	3	4	5
ADDD F6,F4,F6	4	16	17
SD 0(R2),F6	5	18	19
ADDI R1,R1,#8	6	7	8
ADDI R2,R2,#8	7	8	9
SGTI R3,R1,done	8	9	10

Execution Time: 10 clocks